1. Tempo disponibile 120 minuti.
2. Non è possibile consultare appunti, slide, libri, persone, siti web, ecc.
3. Scrivere in modo leggibile, su ogni foglio, nome, cognome e numero di matricola.
4. Le soluzioni agli esercizi che richiedono di progettare un algoritmo devono:
   - spiegare a parole l'algoritmo (se utile, anche con l'aiuto di esempi o disegni),
   - fornire e commentare lo pseudo-codice (indicando il significato delle variabili),
   - calcolare la complessità (con tutti i passaggi matematici necessari),
   - se l'esercizio ammette più soluzioni, a soluzioni computazionalmente più efficienti e/o concettualmente più semplici sono assegnati punteggi maggiori.

1. Calcolare la complessità $T(n)$ del seguente algoritmo `mystery1`:

```
algoritmo mystery1(n: Int) --> Int
    int k = 0
    mystery2(n)
    for i = 1..n
        int j = 1
        while (j<n)
            j = 2*j
            k = k+1
        endwhile
    endfor
    return k

algoritmo mystery2(m: Int) --> Int
    if (m == 1)
        return 2
    else
        return mystery2(m/2)+mystery2(m/2)+mystery2(m/2)+mystery2(m/2)
    endif
```

Soluzione L'algoritmo `mystery1` utilizza `mystery2`. Iniziamo quindi l'analisi da tale secondo algoritmo, che risulta essere ricorsivo con complessità $T'(m)$ caratterizzata dalla seguente equazione di ricorrenza:

$$T'(m) = \begin{cases} 1 & \text{se } m \leq 1 \\ 4T'(\frac{m}{2}) + 1 & \text{altrimenti} \end{cases}$$

viste le 4 chiamate ricorsive con paramentro `m/2`. Applicando il Master Theorem, avendo $a = 4$, $b = 2$ (quindi $\alpha = \frac{\log a}{\log b} = 2$) e $\beta = 1$, otteniamo $T'(m) = O(m^2)$ (considerando il primo caso del teorema). Analizziamo ora `mystery1`. Tutte le operazioni hanno costo costante ad esclusione della invocazione a `mystery2`. Il costrutto `while`, ad ogni sua esecuzione, viene eseguito $O(\log n)$ in quanto l'indice j viene inizializzato a 1 e viene raddoppiato ad ogni ciclo sino a che non diventa maggiore o uguale a n. Il corpo del `for` viene eseguito esattamente n volte; visto che il corpo del `for` include il costrutto `while`, quest'ultimo verrà eseguito n volte, per un costo computazionale complessivo $O(n \log n)$. La complessità di `mystery1` è quindi data dalla somma del costo computazionale della chiamata a `mystery2` con parametro n (costo $T'(n) = O(n^2)$) sommato al costo dell'esecuzione del `for` (costo $O(n \log n)$). Matematicamente, $T(n) = O(n^2) + O(n \log n) = O(n^2)$.

2. Si scriva una procedura ricorsiva che data una lista di interi *monodirezionale* la modifichi sostituendo ogni valore pari con il doppio e replicando due volte gli elementi dispari maggiori di 10.

Esempi:

L1 = 4; 6; 7; 13; 2; 5, allora si ha L1 = 8; 12; 7; 13; 13; 4; 5

L2 = 14; 2; 17; 3; 15, allora si ha L2 = 28; 4; 17; 17; 3; 15; 15

Soluzione Si procede con una scansione ricorsiva partendo dal primo elemento della lista. Se l'elemento analizzato è NULL termina (fine lista). Negli altri casi si analizza il valore e si invoca la funzione sull'elemento successivo. Se il valore è pari viene raddoppiato, se è dispari e maggiore di 10 si aggiunge un nodo alla lista (nota: per duplicare il valore una sola volta si procede sul successore di questo nuovo nodo). Se dispari e minore di 10 si avanza senza modificare il valore.

L'algoritmo fa una scansione della lista ed ha quindi costo lineare.

```
PROCESSA(List nodo) --> void
if (nodo == NULL) then
                return;
        else
                // valore pari: valore raddoppiato
                if (nodo.value % 2 == 0) then
                        nodo.value := nodo.value * 2;
                        PROCESSA(nodo.next);
                // valore dispari maggiore di 10: elemento duplicato
                elseif (nodo.value > 10) then
                        List nd = new List(nodo.value);
                        nd.next := nodo.next;
                        nodo.next = nd;
                        PROCESSA(nd.next);
                else
                        // valore dispari minore o uguale a 10
                        PROCESSA(nodo.next);
                endif
        endif
```

3. Si consideri il seguente gioco che si gioca con carte da poker, ognuna avente un valore compreso fra 1 e 13. Il gioco procede nel seguente modo: si collocano $k \times k$ carte sul tavolo, organizzate secondo una matrice di k righe e k colonne. Si devono prelevare carte secondo il seguente schema: si inizia a prendere una carta a libera scelta dalla prima riga e poi si procede come segue; presa una carta nella riga i e colonna j, si può prendere una carta nella riga $i+1$ e colonna j, oppure nella riga $i+2$ e colonna $j-1$ oppure $j+1$. Il punteggio finale è dato dalla somma dei valori delle carte prelevate dal tavolo. Scrivere un algoritmo che data in input la matrice `C[1..k,1..k]` indicante la distribuzione dei valori delle singole carte inizialmente collocate sul tavolo, calcola il punteggio massimo ottenibile.

Soluzione Sia $C[1..k, 1..k]$ la tabella contenente i valori della carte nelle k righe, ogni riga contenente k carte. Si può utilizzare la programmazione dinamica, considerando dei sottoproblemi $P(i, j)$, con $1 \leq i, j \leq k$, corrispondenti al punteggio massimo che può essere raggiunto iniziando il gioco e prelevando carte fino al prelievo della carta in riga i e colonna j. Tali sottoproblemi possono essere risolti nel seguente modo:

$$P(i,j) = \begin{cases} C[1,j] & \text{se } i = 1 \\ C[1,j] + C[2,j] & \text{se } i = 2 \\ C[i,j] + max\{P(i-2,2), P(i-1,1)\} & \text{se } i > 2 \text{ e } j = 1 \\ C[i,j] + max\{P(i-2,k-1), P(i-1,k)\} & \text{se } i > 2 \text{ e } j = k \\ C[i,j] + max\{P(i-2,j-1), P(i-2,j+1), P(i-1,j)\} & \text{altrimenti} \end{cases}$$

La soluzione al problema sarà quindi data dal punteggio massimo ottenibile prelevando per ultima una della carte nell'ultima riga, quindi $max\{P(i, j) \mid i = k, 1 \leq j \leq k\}$.

Possiamo quindi utilizzare il seguente algoritmo che fa uso della matrice `S[1..k,1..k]` che verrà riempita in modo tale che `S[i,j]` conterrà $P(i,j)$. Dopo aver riempito la matrice si restituirà il valore massimo dell'ultima riga. Possiamo quindi utilizzare il seguente algoritmo (in cui, per comodità, assumiamo $k \geq 2$):

```
algoritmo GiocoCarte(C: Int[1..k,1..k]) --> Int

   //dichiarazione variabili ausiliarie
   Int S[1..k,1..k], valMax=0

   //inizializzazione prime due righe della tabella
   for j=1..k
      S[1,j]=C[1,j]
      S[2,j]=C[1,j]+C[2,j]
   endfor

   //riempimento restanti righe
   for i=3..k
      S[i,1]=C[i,1]+max{S[i-2,2],S[i-1,1]}
      S[i,k]=C[i,k]+max{S[i-2,k-1],S[i-1,k]}
      for j=2..k-1
         S[i,1]=C[i,j]+max{S[i-2,j-1],S[i-2,j+1],S[i-1,j]}
      endfor
   endfor

   //ricerca del massimo
   for j=1..k
      if (S[k,j]>valMax) valMax=S[k,j]
   endfor

   return valMax
```

La complessità di tale algoritmo deriva dal numero di operazioni eseguite dai cicli, ovvero $T(k) = O(k^2)$.

4. Progettare un algoritmo che, dato un grafo non orientato $G = (V, E)$, tre vertici v_1, v_2 e v_3 e un intero k, verifica se esiste un cammino di lunghezza inferiore a k che va da v_1 a v_3 passando per v_2.

Soluzione Tale problema consiste nel verificare che la somma delle distanze da v_1 a v_2 e da v_2 a v_3 sia inferiore a k. Essendo il grafo non orientato, possiamo semplicemente effettuare una BFS a partire da v_2 per calcolare la distanza tra tale nodo ed i nodi v_1 e v_3, per poi verificare che la somma di queste due distanze sia inferiore a k. Per comodità, si inizializzano le distanze di v_1 e v_3 ad infinito, in modo tale che se anche uno solo dei due nodi risulta non raggiungibile, la relativa distanza rimarrà uguale ad infinito e quindi l'algoritmo restuirà `false`.

```
algoritmo verificaCammino(Graph G=(V,E), Node v1, Node v2, Node v3, Int k) --> Boolean

   //inizializzazione distanze di v1 e v2
   v1.dist = +INFINITO
   v3.dist = +INFINITO

   //visita in ampiezza
   for each v in V do v.mark = false
   F = new Queue()
   F.enqueue(v2)
   v2.mark = true
   v2.dist = 0
   while (not F.isEmpty) do
      u = F.dequeue()
      for each v adiacente a u do
```

```
            if (not v.mark) then
                v.mark = true
                v.dist = u.dist+1
                F.enqueue(v)
            endif
        endfor
    endwhile

    //verifica esistenza cammino di lunghezza inferiore a k
    return (v1.dist+v3.dist < k)
```

La complessità di tale algoritmo coincide con la complessità dell'algoritmo BFS, ovvero $T(n, m) = O(n + m)$ con n numero di nodi, m numero di archi del grafo in input e assumendo implementazione del grafo tramite liste di adiacenze.

The End.